32

USING THE BASICX MICROCONTROLLER

Microcontrollers are fast becoming a favorite method for endowing robots with smarts. In fact, they're a robot builder's dream come true. Microcontrollers are single-chip computers complete with their own input/output ports and even memory. The typical cost of a microcontroller is from $5 to $15 and most can be programmed using the software on your PC. Once programmed, the microcontroller is disconnected from the PC and operates on its own. Microcontrollers are power misers too. Nearly all have simple power requirements (usually just 3.3 or 5 volts) and require just a few milliamps for their own operation, even when running at speeds of 5 or 10 megahertz.

Microcontrollers are available in two basic flavors: *low-level programmable* and *embedded-language programmable.* As we noted in Chapter 28, these loosely defined terms relate to the programming of the controller. Both kinds of microcontroller are fully programmable, but one contains a kind of built-in operating system that allows it to be programmed with a higher-level language, such as Basic.

Let's talk about low-level microcontrollers first. You program these with assembly language or C, using your PC as a host development system. Assembly language seems somewhat arcane to newcomers, but the language offers full control over the internal workings of the microcontroller. Unfortunately, there's no standard when it comes to assembly languages.

Popular alternatives to these low-level programmable microcontrollers are products that have a built-in programming interface, such as the Basic Stamp from Parallax or the OOPic from Savage Industries. These controllers support a high-level programming language—typically Basic—that is permanently embedded within the chip. Using your PC as a develop-

ment platform, you write software for the microcontroller using a custom program editor. The software is then compiled to a series of *tokens* or *bytecodes* and then downloaded to the microcontroller.

Joining the ranks of powerful embedded-language programmable microcontrollers is the BasicX-24, by NetMedia, a company founded by the creator of the popular LANtastic networking software (which sold some 10 million copies). The BasicX-24 is actually a member of a family of microcontrollers from NetMedia that also includes the less expensive (but network-capable) BasicX-01. However, all things considered, the BasicX-24 is perhaps the most versatile, so this chapter will focus on it exclusively.

Inside the BasicX-24 Microcontroller

A selling point of the BasicX-24 (which we'll refer to as the BX-24 from here on) is that it is pin-for-pin compatible with Parallax's Basic Stamp II. That is, the functions of all 24 pins of the BX-24 replicate the functions of the Basic Stamp II, including power and ground connections. It's important to note, however, that the BX-24 is not a Stamp "clone." The two microcontrollers don't share the same programming languages, so programs written for one will not work on the other. Additionally, the BX-24 has several additional features not found in the Basic Stamp II, such as built-in analog-to-digital conversion and 32K of EEPROM memory.

Fig. 32.1 shows the BX-24 "chip," which (like the Basic Stamp) is actually several integrated circuits on a small circuit board. The layout of the pins on the BX-24 is identical to that of any standard-sized 24-pin IC, so it will plug into a regular 24-pin socket. Additional plated-through holes are provided on either end of the BX-24 board, making it just slightly longer than the Basic Stamp II. These holes provide connections to additional input/output lines provided on the BX-24. I'll get to those in a bit.

The BX-24 directly supports 16 input/output (I/O) lines, the same number as the Basic Stamp II. For each I/O line, or pin, you can change the direction from an input or an output. When an I/O line is an output, you can individually control the value of the pin, either 0 (logic LOW) or 1 (logic HIGH). When an I/O line is an input, you can read a digital or analog value from a TTL-compatible device connected to the BX-24. Eight of the 16 I/O lines can be used for analog connections. The BX-24 incorporates its own built-in 10-bit analog-to-digital converter (ADC). Under software control, you can indicate which of the 8 input lines is to be read.

Three of the plated-through holes of the BX-24 serve as optional I/O and are programmatically referred to as pins 25, 26, and 27. This makes a total of 19 input/output pins. The remaining plated-through holes provide a way to connect to the chip's serial peripheral interface, or SPI, lines. I do not recommend that you connect to these lines unless you're familiar with SPI interfaces, especially since the BX-24's EEPROM is controlled by these same I/O lines.

A nice feature of the BX-24 is its two LEDs: one red and one green. The green LED is normally used to indicate that the chip is powered on, but you can individually control both LEDs from your own programs. You might use the LEDs as status indicators, for example. The LEDs share two of the additional plated-through hole connectors on the BX-24.

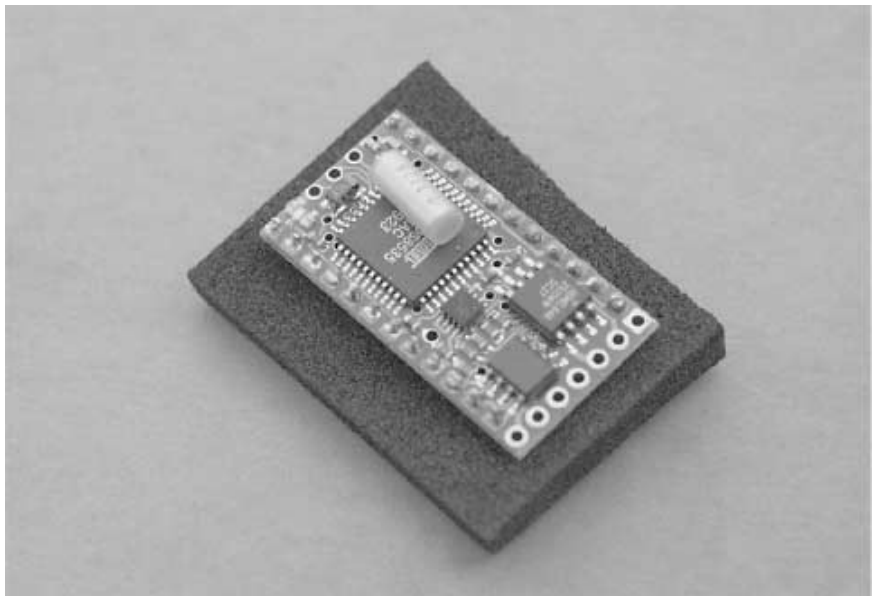

FIGURE 32.1 **The BasicX-24 consists of surface-mount integrated circuits on a small circuit board. The BX-24 circuit board has the same dimensions as a standard 24-pin IC.**

The BX-24 board comes with its own five-volt voltage regulator, which provides enough operating current for all the components on the board, plus several LEDs or logic ICs. If you plan on using the BX-24 to operate a robot, you'll want to provide a separate power supply of adequate current rating to the other components of the robot. You should not rely on the BX-24's on-board regulator for this task.

Pinout Diagram for the BX-24

Fig. 32.2 shows the pinout diagram of the BX-24 as well as the functions of its 24 pins. Of main interest are the following:

Pin 24. This is the unregulated power input. Apply an unregulated DC voltage of 5.5 to 15 volts here. The onboard regulator will provide a stable 5 vdc input for the BX-24 circuitry.
Pin 23, 4. This is the ground. You can use either or both of these pins when connecting to other circuitry.
Pin 21. This is for 5 vdc input. Instead of using pin 24 for power, you may directly apply *regulated* 5 vdc to this pin. Or, if power is applied through pin 24, pin 21 serves as a convenient source of regulated 5 vdc power. The voltage regulator on the BX-24

can supply approximately 70 mA of total additional current, either through this pin or through the I/O pins described next.

Pins 5 through 11. This is I/O Port C, one of two eight-bit ports on the BX-24. Pin 12 serves "double duty" as an input capture pin, which can be used for very accurate timing. Pin 11 serves double duty as an external interrupt. With the appropriate programming, the BX-24 can be commanded to automatically run certain code when this line goes HIGH.

Pins 13 through 20. This is Port D, the second of two eight-bit ports on the BX-24. All of the pins in this port serve double duty as analog-to-digital conversion inputs. That is, in addition to on/off (1 and 0) digital inputs and outputs, these pins can accept analog inputs. The range of the analog inputs spans 0 to 5 volts.

Pins 25 and 26 are additional I/O lines that are available if you solder connections directly to the BX-24 chip (as such as they are not strictly "pins," but we'll treat them as if they were). Access to pins 25 and 26 are provided via plated-through holes, which can be connected to wires or pin headers. These two pins share I/O with the on-board red and green LEDs.

Pin 27 serves as the output capture I/O line. As with pins 25 and pin 26, this pin is available if you solder directly to the BX-24 chip.

Programming the BX-24

To program the BX-24 you need to purchase the BasicX-24 developer's kit, which contains one BX-24, a programming cable, a power supply, a "carrier board" (see Fig. 32.3), and programming software on CD-ROM. You plug the BX-24 into the carrier board, which has a 24-pin socket and empty solder pads that you can use to add your own circuitry. The programming cable connects between the carrier board and a serial port on your PC. The power supply is the "wall wart" variety and provides about 12–16 vdc.

The BX-24 uses a proprietary programming environment, consisting of an editor and a download console, which also serves double duty as a terminal for data sent from the

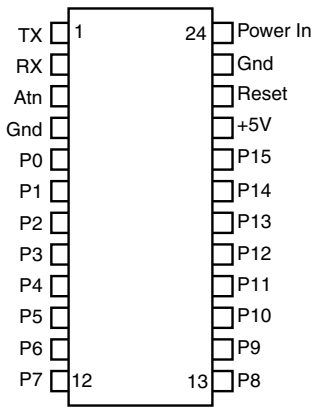

FIGURE 32.2 **Pinout diagram of the BasicX-24 chip. Note that several of the pins serve double duty (as explained in the text).**

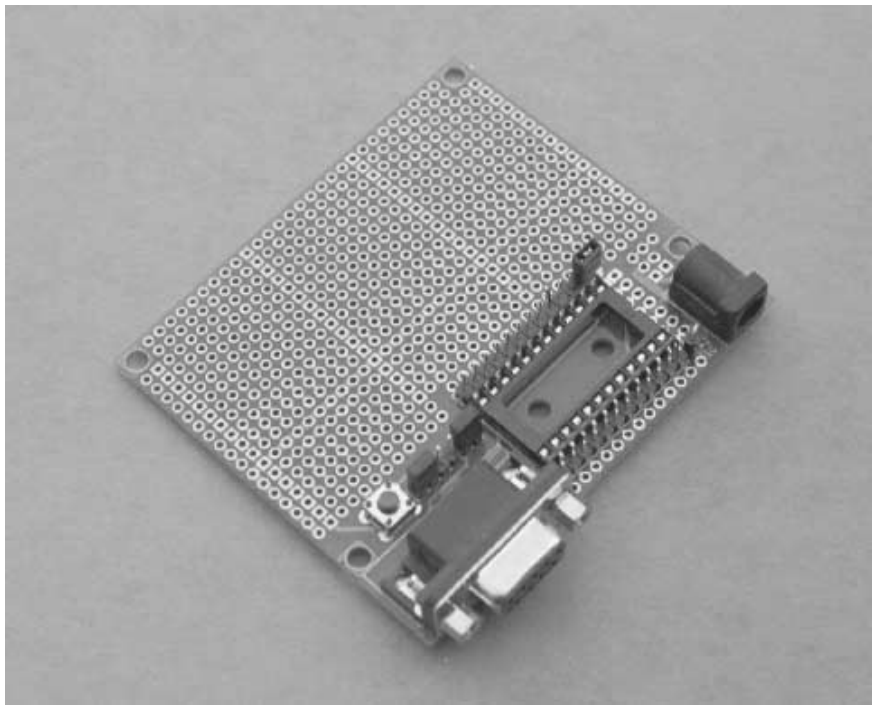

FIGURE 32.3 **The easiest way to experiment with the BX-24 is to use the carrier board that is included as part of the BX-24 developer's kit. The carrier board includes a DB-9 connector for hooking the system up to a PC for programming.**

microcontroller. The program editor, shown in Fig. 32.4, supports the BasicX language, which is a subset of Microsoft Visual Basic. Don't expect all Visual Basic commands to be available in BasicX, however. BasicX supports the same general syntax as Visual Basic, and many of the same data types (bytes, integers, strings, and so forth).

If you're familiar with Visual Basic then you should feel right at home with BasicX. The BasicX language supports the usual control structures, such as *If…End If, While…Wend, For—Next,* and *Select…Case.* Your BasicX programs can be subroutines, and you can call those subroutines from anywhere in the program.

Depending on how you've used Visual Basic, however, you may discover that BasicX is far less forgiving of certain programming habits. BasicX uses a "strict" data-typing syntax that requires you to use the *Dim* statement—or one of its variations, such as *Const*—to define each variable before it is used. With the *Dim* statement you must also indicate the variable type, such as *Byte* or *String.*

Modern versions of Visual Basic support a special type of variable called the *variant.* Variants can hold most any kind of data, which allows you to freely "mix and match" data types, such as adding an integer to a string (i.e., adding the number one to the name

FIGURE 32.4 **Use the BasicX program editor to create, edit, compile, and (optionally) download programs for the BX-24.**

"Smith" to get "Smith1"). Apart from the danger that you will introduce bugs by mixing data types, variants consume a lot of memory. They also tend to slow down execution speed, since it must determine the type of variable each time it is accessed.

Visual Basic provides the variant feature because memory is abundant on PC systems, and—at least with the latest machines—processor speed is fairly fast. Conversely, memory in a microcontroller must be carefully rationed. The BX-24 supports 400 bytes (that's bytes, not megabytes or even kilobytes) of RAM memory to store data. For a microcontroller, that's actually a copious amount of memory! (By the way, if you're wondering, your programs are stored separately in a 32K block of EEPROM, which is enough for some 8000 instructions. You'll be hard-pressed to create programs that large for your robot.)

When using BasicX, you must be constantly aware of the data type being stored in each variable. If you need to manipulate two variables that contain different types of data, you must remember to use the various data conversion commands that BasicX supports. This is perhaps one of the most frustrating aspects of BasicX programming for newcomers.

A particularly nice feature of the BasicX editor is that it allows you to build "projects" consisting of multiple files. This allows you, for instance, to build a library of commonly used programming functions that you may regularly use in your robotics work. When building a new program for the BX-24, you create a new project and then include any constituent files. This saves you from having to manually cut and paste commonly used code to make one big program file.

Advanced programmers will appreciate the ability to work with real arrays in the BX-24 environment. You can create arrays of any data type except strings or other arrays. You can then reference the elements of the array using an index number. This feature makes it handy to manipulate such things as data streams, where you want to store a series of bytes in one compact package.

Before you can send your programs to the BX-24 chip they must be compiled. This is done in the BasicX editor by choosing the *Compile* command from the *Compile* menu.

Compiling can take a while on slower machines, so be patient. Syntax errors are flagged, and if they are found, compiling stops. When you have successfully compiled the program it can then be downloaded to the BX-24 chip. This can be done from the BasicX editor or from the download console. After the program has been successfully compiled, it can be redownloaded any number of times. It does not need to be recompiled before each download.

Multitasking with the BX-24

One of the more valuable uses subroutines provide is the ability to create multitasking programs. Multitasking is a built-in feature of the BasicX operating system. In most instances, the multitasking is "preemptive," meaning that the BasicX operating system forces the BX-24 microcontroller to "time-slice" between each multitasked subroutine. Each slice is given 1/512 of a second, more than enough to complete over a hundred instructions before moving on to the next subroutine. (The BX-24 processes some 65,000 instructions per second, or approximately 127 instructions per time-slice.) A few of the commands supported in the BasicX system suspend multitasking because they are sensitive to timing. These include such commands as *InputCapture* (explained later in this chapter), which accurately measures the duration of signals received by the BX-24.

While multitasking is a powerful feature of the BX-24, it's not always easy to implement. For each subroutine that you wish to multitask you must manually calculate the amount of RAM needed to hold data for that subroutine while the system switches. This calculation is necessary so sufficient "stack space" is allocated to hold the data as the BX-24 services each task. If you underestimate the RAM requirements, your program won't work properly; if you overestimate the requirements, you waste precious memory.

BasicX Functions for Robotics

The BX-24 is a general-purpose microcontroller, so many of its built-in features are geared toward any typical personal or commercial microcontroller application. Still, a number of features of the BasicX programming language lend themselves for use in robotics. These features are implemented as functions added to the BasicX language. To use a feature, you merely include it in your program along with any necessary command parameters.

Note: Several of these functions require you to use version 1.45 or later of the BasicX compiler. If you're already a BX-24 owner, you'll also need to make sure that your chip has the latest BasicX operating system firmware embedded into it. Check the BasicX site (www.basicx.com) for details.

REAL-TIME CLOCK

The BX-24 contains its own real-time clock (RTC), accurate to within several seconds per day. You must set the correct time whenever you power up the BX-24, but once the time is set, you can use the RTC to measure events. For example, you can write a robot program

that accurately marks the time it takes to travel from one room to another. The RTC is also handy for data logging, which allows your robot to roam around the house or yard and store data from its sensors. Coupled with the BX-24's ability to optionally store data in EEPROM, the data log will survive even if power is removed to the chip.

GETADC AND PUTDAC

As mentioned earlier, the BX-24 has its own eight-channel, 10-bit ADC. With the *GetADC* function, you can read a voltage level on any of eight I/O pins and correlate that voltage level with a binary number (from 0 to 1023). Conversely, you can use the *PutDAC* function to output a pulse train that will mimic a variable voltage.

SHIFTIN AND SHIFTOUT

With *ShiftIn* you can receive a series of bits on a single I/O pin and convert them to a single byte in a variable. *ShiftOut* does the inverse, converting a byte into a series of bits. Both functions allow you to specify an I/O pin to be used as the data source and another I/O pin for the clock. The BasicX software automatically triggers the clock pin for each bit received or sent. The *ShiftIn* and *ShiftOut* functions are particularly handy when you are using serially based components, which allow you to interface with devices using only two I/O lines.

OPENCOM

The BX-24 supports as many serial ports as you have available I/O pins. With *OpenCom* you can establish serial communications with other BX-24 chips or any other device that supports serial data transfer. One common use for *OpenCom* is to establish a link from the BX-24 chip back to the download window of your PC. This window can serve as a terminal for debugging and other monitoring tasks.

PULSEIN AND PULSEOUT

The *PulseIn* function waits for the level at a given I/O pin to change state. One practical application of this feature is to activate some function on your robot when a critical button is pressed. *PulseOut* sends a pulse of a certain duration (in 1.085 microsecond units) out a given I/O pin. *PulseOut* is one of the most commonly used functions and is used to blink LEDs, trigger sonar pings, and command servo motors to move to a new location. Note that both *PulseIn* and *PulseOut* turn off the task-switching feature of the BX-24. Several other BasicX functions behave in the same way because they literally "take over" the chip. Because these functions hog processor time, both can also cause errors in the real-time clock.

INPUTCAPTURE

Somewhat akin to *PulseIn, InputCapture* watches for signal transition on a specific I/O pin of the BX-24. *InputCapture* can time the duration of these transitions, thereby giving you a "snapshot" of a digital pulse train, including how long each pulse lasted. One application of *InputCapture* is watching for and decoding the serial signals from an infrared remote control.

PLAYSOUND

The *PlaySound* function outputs a waveform that, when connected to an amplifier via a decoupling capacitor, allows you to play previously sampled sound that has been stored in the EEPROM. You can play back sounds at various sampling rates and control the number of times the sound is repeated. The repeat function is a handy way to stretch a relatively short sound sample into a longer one—for example, the "chug-chug" of a machine motor or a series of blips.

ADDITIONAL USEFUL FUNCTIONS FOR ROBOTICS

In addition to BX-24's built-in functions, you can access many of the internal hardware registers of the BX-24 chip. The BX-24 is based on the Atmel AT90S8535 microcontroller (download the data sheet for the '8535 to learn more about the internals of this powerful chip). By controlling the hardware registers of the BX-24 you can program features that the BasicX language itself does not directly support. For example, by setting a few registers for Timer1 (one of three timers in the Atmel '8535), you can produce dual pulse width modulated (PWM) signals, which are useful for controlling the speed of DC motors. In a practical circuit, you will need to interface the two PWM outputs of the BX-24 to a suitable transistor or H-bridge circuit in order to provide enough drive current to run the motors.

Working directly with the hardware registers of the BX-24 is not for the feint of heart, however. If you want to try this technique, first study the Atmel AT90S8535 data sheet and learn how the registers of the chip work. It's entirely possible to set the registers in a way that will crash the chip, rendering it inoperative (of course, you can always reset the BX-24 and try again with a new program).

A Sample BX-24 Program

Constructing a BX-24 program involves at least one subroutine, called Main, and one or more BasicX commands. In the following program example, the BX-24 flashes its red and green LEDs on and off several times each second.

LISTING 32.1

```
Sub Main()
' BX-24 LED demonstration.
Const GreenLED As Byte = 26
Const RedLED As Byte = 25
Const LEDon As Byte = 0
Const LEDoff As Byte = 1

Do
    '  Red pulse.
    Call PutPin(RedLED, LEDon)
    Call Delay(0.07)
    Call PutPin(RedLED, LEDoff)

    Call Delay(0.07)

    ' Green pulse.
```

```
      Call PutPin(GreenLED, LEDon)
      Call Delay(0.07)
      Call PutPin(GreenLED, LEDoff)

      Call Delay(0.07)
Loop
End Sub
```

Here's how the program works. The following commands,

```
Sub Main()
  ...
End Sub
```

form the main subroutine that is automatically executed when the BX-24 is first turned on or when it is reset. You can have additional subroutines in the program, each with a different name, but at a minimum you need one subroutine called *Main* to get things started:

```
Const GreenLED As Byte = 26
Const RedLED As Byte = 25
Const LEDon As Byte = 0
Const LEDoff As Byte = 1
```

These lines define four constants, using the *Const* statement (similar to *Dim*). *Const* stands for "constant" and represents a variable that will never be changed again in the program. In this example, each *Const* statement defines three things:

The name of the variable, such as *GreenLED* or *LEDon.*
The type of variable (how many bits it requires). In all four instances the variables are of type *Byte* and each requires eight bits
The value of each variable. For example, *GreenLED* is assigned the value 26; *LEDoff* is assigned the value 0.

All four constants are used elsewhere in the program, and they serve as a convenient way to change values should that ever be necessary. The statements,

```
Do
    ...
Loop
```

set up an "infinite loop." That is, the loop repeats for as long as power is applied to the BX-24 (or until the chip is reset). Without the *Do...Loop* statements the commands in the program would execute just once. The loop provides a simple way to repeat the commands indefinitely:

```
' Red pulse.
Call PutPin(RedLED, LEDon)
Call Delay(0.07)
Call PutPin(RedLED, LEDoff)
Call Delay(0.07)
```

Each BasicX function, such as *PutPin,* is preceded by an optional *Call* statement. This tells the BasicX operating system to perform the named function. The *PutPin* function, called

twice in this example, changes the state of a specified I/O line. Note the use of the constants. The syntax for *PutPin* is as follows:

```
PutPin (PinNumber; Value)
```

where *PinNumber* is the number of the pin you want to use (e.g., pin 25 for the red LED), and *Value* is either 1 for on (or logical HIGH) or 0 for off (or logical LOW).

The *Delay* function causes the BX-24 to pause a brief while, in this case 70 milliseconds. *Delay* is called twice, so there is a period of time between the on/off flashing of each LED:

```
' Green pulse.
Call PutPin(GreenLED, LEDon)
Call Delay(0.07)
Call PutPin(GreenLED, LEDoff)
Call Delay(0.07)
```

The process is repeated for the green LED.

Controlling RC Servos with the BX-24

You can easily control RC servos with the BX-24 using a few simple statements. While there is no built-in "servo command" as there is with the OOPic microcontroller (see Chapter 33), the procedure is nevertheless very easy to do in the BX-24. Here's a basic program that places a servo connected to pin 20 of the BX-24 at its approximate mid-point position. (I say "approximate" because the mechanics of RC servos can differ between makes, models, and even individual units):

```
Sub Main
Do
    Call PulseOut(20, 1.5E-3, 1)
    Call Delay(0.02)
Loop
End Sub
```

The program continuously runs because it's within an infinite *Do* loop. The *PulseOut* statement sends a short 1.5-millisecond (ms) HIGH pulse to pin 20. The *Delay* statement causes the BX-24 to wait 20 milliseconds before the loop is repeated all over again. With a delay of 20 milliseconds, the loop will repeat 50 times a second (50 * 20 milliseconds < 1000 milliseconds, or one second).

Note the optional use of scientific notation for the second parameter of *PulseOut*. Using the value 0.0015 would yield the same result. You should be aware that the BX-24 supports two versions of the *PulseOut* statement: a float version and an integer version:

The *float* version is used with floating-point numbers, that is, numbers that have a decimal point.

The *integer* version is used with integers, that is, whole numbers only.

The BX-24 compiler automatically determines which version to use based on the data format of the second parameter of the *PulseOut* statement. If you use

```
Call PulseOut(20, 20, 1)
```

it tells the BX-24 you want to send a pulse of 20 "units." A unit is 1.085 microseconds long; 20 units would produce a very short pulse of only 21.7 microseconds. To continue working in more convenient milliseconds, be sure to use the decimal point:

```
Call PulseOut(20, 0.020, 1)
```

This creates a pulse of 20 milliseconds in length.

Listing 32.2 shows a more elaborate servo control program and is based on an application note provided on the BasicX Web site. This program allows you to specify the position of the servo shaft as a value from 0 to 100, which makes it easier for you to use.

LISTING 32.2

```
Const ServoPin As Byte = 20
Const RefreshPeriod As Single = 0.02
Const NSteps As Integer = 100
Dim SetPosition As Byte
Dim Position As Single, PulseWidth As Single

Sub Main ()
' Moves a servo by sending a single pulse.
' Insert position as a value from 0 to 100
SetPosition = 50       ' move to mid-point

Position = CSng(SetPosition) / CSng(NSteps)
Do
      ' Translate position to pulse width, from 1.0 to 2.0 ms
      PulseWidth = 0.001 + (0.001 * Position)

      ' Generate a high-going pulse on the servo pin
      Call PulseOut(ServoPin, PulseWidth, 1)
      Call Delay(RefreshPeriod)
Loop
End Sub
```

The five lines at the beginning of the program set up all the variables that are used. The line

```
Const ServoPin As Byte = 20
```

creates a byte-sized constant and also defines the value of the constant as pin 20. Because it is a constant, the value assigned to *ServoPin* cannot be changed elsewhere in the program. Similarly, the lines

```
Const RefreshPeriod As Single = 0.02
Const NSteps As Integer = 100
```

create the constants *RefreshPeriod* and *NSteps*. *RefreshPeriod* is a single-precision floating-point number, meaning that it can accept numbers to the right of the decimal point. *Nsteps* is an integer and can accept values from Ω32768 to 32767.

The main body of the program begins with *Sub Main*. The statement

```
SetPosition = 50
```

sets the desired position of the servo relative to the total number of steps defined in *NSteps* (in the case of our example, 100). Therefore, a *SetPosition* of 50 will move the servo to its approximate midpoint. The line

```
Position = CSng(SetPosition) / CSng(NSteps)
```

produces a value from 0.0 to 1.0, depending on the number you used for *SetPosition.* With a value of 50, the *Position* variable will contain 0.5. The *Position* variable is then used within the *Do* loop that follows. Within this loop are the following statements:

```
PulseWidth = 0.001 + (0.001 * Position)
Call PulseOut(ServoPin, PulseWidth, 1)
Call Delay(RefreshPeriod)
```

The first statement sets the pulse width, which is between 1.0 and 2.0 milliseconds. The *PulseOut* statement sends the pulse through the indicated servo pin (the third parameter, 1, specifies that the pulse is positive-going, or HIGH). Finally, the *Delay* statement delays the BX-24 for the *RefreshPeriod,* in this case 20 milliseconds (0.02 seconds).

Reading Button Inputs and Controlling Outputs

A common robotics application is reading an input, such as a button, and controlling an output, such as an LED, motor, or other real-world device. Listing 32.3 shows some simple code that reads the value of a momentary push button switch connected to I/O pin 20. The switch is connected in a circuit, which is shown in Fig. 32.5, so when the switch is open, the BX-24 will register a 0 (LOW), and when it's closed the BX-24 will register a 1 (HIGH).

The instantaneous value of the switch is indicated in the LED. The LED will be off when the switch is open and on when it is closed.

LISTING 32.3

```
Sub Main()
Const InputPin As Byte = 20
Const LED As Byte = 26
Dim State as Byte
Sub Main()
Do
         ' Read I/O pin 20
         State = GetPin(InputPin)
         ' Copy it to the LED
         Call PutPin(LED, State)
```

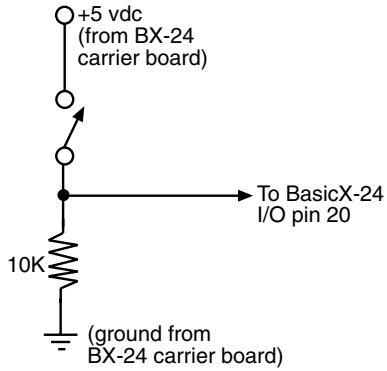

FIGURE 32.5 **Wire the switch so it connects to the V (pin 21, *not* pin 24) of the BX-24. The resistors are added for safety.**

```
Loop
End Sub
```

Now let's see how the program works. The lines,

```
Const InputPin As Byte = 20
Const LED As Byte = 26
Dim State as Byte
```

set the constant *InputPin* as I/O pin 20, and the constant LED as I/O pin 26. (Recall that one of the BX-24's on-board LEDs—the green one, by the way—is connected to I/O pin 26.) Finally, the variable *State* is defined as type *Byte:*

```
Do
       ' Read I/O pin 20
       State = GetPin(InputPin)
       ' Copy it to the LED
       Call PutPin(LED, State)
Loop
```

The *Do* loop repeats the program over and over. The *GetPin* statement gets the current value of pin 20, which will either be LOW (0) or HIGH (1). The companion *PutPin* statement merely copies the state of the input pin to the LED. If the switch is open, the LED is off; if it's closed, the LED is on.

Additional BX-24 Examples

So far we've just scratched the surface of the BX-24's capabilities. But fear not: throughout this book are several real-world examples of BX-24 being using in robotic applications. For instance, in Chapter 41 you'll learn how to use the BX-24 to interface to a sophisticated accelerometer sensor. In addition, you can find several application notes for the BX-24 (and its "sister" microcontrollers, such as the BX-01) on the BasicX Web page (www.basicx.com).

From Here

To learn more about…	*Read*
Stepper motors	Chapter 19, "Working with Stepper Motors"
How servo motors work	Chapter 20, "Working with Servo Motors"
Different approaches for adding brains to your robot	Chapter 28, "An Overview of Robot 'Brains'"
Connecting the OOPic microcontroller to sensors and other electronics	Chapter 29, "Interfacing with Computers and Microcontrollers"